Working with DB2 UDB data

Presented by DB2 Developer Domain

`http://www7b.software.ibm.com/dmdd/`

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Introduction

What this tutorial is about

This tutorial introduces you to Structured Query Language (SQL), and helps to give you a good understanding of how DB2 Universal Database uses SQL to define database objects (such as tables, views, or indexes) and to manipulate data in a relational database.

This tutorial is the fourth in a series of six tutorials that you can use to help prepare for the DB2 V 8.1 Family Fundamentals Certification (Exam 700). The material in this tutorial primarily covers the objectives in Section 4 of the exam, which is entitled "Working with DB2 UDB Data." You can view these objectives at: *http://www.ibm.com/certify/tests/obj700.shtml*.

DB2 installation is not covered in this tutorial. If you haven't already done so, you can download and install a copy of *IBM DB2 Universal Database*, Enterprise Server Edition. Installing DB2 will help you understand many of the concepts that are tested on the DB2 UDB V8.1 Family Fundamentals Certification exam. The installation process is documented in the Quick Beginnings books, which can be found at the *DB2 Technical Support* Web site under the Technical Information heading.

In this tutorial, you will learn:

- The fundamentals of SQL, with a focus on SQL language elements
- How Data Control Language (DCL) is used to provide access control to database objects
- How Data Definition Language (DDL) is used to create, modify, or drop database objects
- How Data Manipulation Language (DML) is used to select, insert, update, or delete data
- How to create and call SQL procedures from the command line

Conventions used in this tutorial

The following text highlighting conventions are used in this tutorial:

- `Monospaced` text is used for SQL statements. `UPPERCASE` text identifies SQL keywords, and `lowercase` text identifies user-supplied values in example code.
- Except in the code, database object names are given in uppercase characters, and table column names are given in mixed case.
- *Italic* text is used when introducing a new term, or to identify a parameter variable.

All of the examples in this tutorial are based on the SAMPLE database, which comes with DB2 Universal Database. As noted in the last panel, you can download a trial version of IBM DB2 Universal Database; however, the examples can be understood without access to the product, as sample output is provided in most cases.

About the author

Roman B. Melnyk, PhD, is a senior member of the DB2 Information Development team, specializing in database administration, DB2 utilities, and SQL. During more than eight years at IBM, Roman has written numerous DB2 books and other related materials. Roman recently coauthored *DB2: The Complete Reference* (Osborne/McGraw-Hill, 2001), *DB2 Fundamentals Certification for Dummies* (Hungry Minds, 2001), and *DB2 for Dummies* (IDG Books, 2000). You can reach him at *roman_b_melnyk@hotmail.com*.

Section 2. Structured Query Language (SQL)

The parts of speech of SQL

SQL is a language used to define and manipulate database objects. SQL is the language that you use to define a database table, insert data into the table, change the data in the table, and retrieve data from the table. Like all languages, SQL has a defined syntax and set of language elements.

Most SQL statements contain one or more of the following language elements:

- **Characters:** Single-byte *characters* can be a letter (A-Z, a-z, $, #, and @, or a member of an extended character set), a digit (0-9), or a special character (including the comma, asterisk, plus sign, percent sign, ampersand, and several others).
- **Tokens:** A *token* is a sequence of one or more characters. A token cannot contain blank characters unless it is a delimited identifier (one or more characters enclosed by double quotation marks) or a string constant.
- **Identifiers:** An SQL *identifier* is a token that is used to form a name.
- **Data types:** The *data type* of a value determines how DB2 interprets that value. DB2 supports a large number of built-in data types, and also supports user-defined data types (UDTs). For information about DB2's built-in data types, see Data types on page 5 .
- **Constants:** A *constant* specifies a value. Constants are classified as character, graphic, or hexadecimal *string constants,* or integer, decimal, or floating-point *numeric constants.*
- **Functions:** A *function* is a relationship between a set of input data values and a set of result values. Database functions can be either built in or user defined. The argument of a *column* function is a collection of like values; a column function returns a single value. Examples of built-in column functions are `AVG`, `MIN`, `MAX`, and `SUM`. The arguments of a *scalar* function are individual scalar values; a scalar function returns a single value. Examples of built-in scalar functions are `CHAR`, `DATE`, `LCASE`, and `SUBSTR`.
- **Expressions:** An *expression* specifies a value. There are string expressions, arithmetic expressions, and case expressions, which can be used to specify a particular result based on the evaluation of one or more conditions.
- **Predicates:** A *predicate* specifies a condition that is true, false, or unknown about a given row or group. There are subtypes:
  - A *basic predicate* compares two values (for example, *x > y*).
  - The *BETWEEN* predicate compares a value with a range of values.

- The *EXISTS* predicate tests for the existence of certain rows.
- The *IN* predicate compares one or more values with a collection of values.
- The *LIKE* predicate searches for strings that have a certain pattern.
- The *NULL* predicate tests for null values.

Data types

The built-in data types can be classified as *numeric, character string, graphic string, binary string,* or *datetime.* There is also a special data type named DATALINK. A *DATALINK value* contains a logical reference to a file stored outside of the database.

Numeric data types include SMALLINT, INTEGER, BIGINT, DECIMAL(*p*,*s*), REAL, and DOUBLE. All numbers have a sign and a precision. The precision is the number of bits or digits excluding the sign. The sign is considered positive if the value of a number is zero or more.

- **Small integer, SMALLINT:** A small integer is a two-byte integer with a precision of 5 digits. The range of small integers is -32,768 to 32,767.

- **Large integer, INTEGER, or INT:** A large integer is a four-byte integer with a precision of 10 digits. The range of large integers is -2,147,483,648 to 2,147,483,647.

- **Big integer, BIGINT:** A big integer is an eight-byte integer with a precision of 19 digits. The range of big integers is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

- **Decimal, DECIMAL(*p,s*), DEC(*p,s*), NUMERIC(*p,s*), or NUM(*p,s*):** A decimal value is a packed decimal number with an implicit decimal point. Packed decimal numbers are stored in a variation of binary-coded decimal (BCD) notation. The position of the decimal point depends on the precision *(p)* and the scale *(s)* of the number. The scale, which is the number of digits in the fractional part of the number, cannot be negative or greater than the precision. The maximum precision is 31 digits. The range of decimals is -10**31+1 to 10**31-1.

- **Single-precision floating-point, REAL:** A single-precision floating-point number is a 32-bit approximation of a real number. The number can be zero or can range from -3.402E+38 to -1.175E-37, or from 1.175E-37 to 3.402E+38.

- **Double-precision floating-point, DOUBLE, DOUBLE PRECISION, or FLOAT:** A double-precision floating-point number is a 64-bit approximation of a real number. The number can be zero or can range from -1.79769E+308 to -2.225E-307, or from 2.225E-307 to 1.79769E+308.

A character string is a sequence of bytes. Character strings include fixed-length character strings of type CHAR(*n*), and varying-length character strings of type VARCHAR(*n*), LONG VARCHAR, or CLOB(*n*). The length of the string is the number of bytes in the sequence.

- **Fixed-length character string, CHARACTER(*n*) or CHAR(*n*):** A fixed-length character string is between 1 and 254 bytes long. If the length is not specified, it is assumed to be 1 byte.

- **Varying-length character string, VARCHAR(*n*), CHARACTER VARYING(*n*), or CHAR VARYING(*n*):** VARCHAR(*n*) type strings are varying-length character strings that can be up to 32,672 bytes long.

- **LONG VARCHAR:** LONG VARCHAR type strings are varying-length character strings that can be up to 32,700 bytes long.

- **Character Large Object String, CLOB(*n*[K|M|G]):** A CLOB is a varying-length character string that can be up to 2,147,483,647 bytes long. If only *n* is specified, the value of *n* is the maximum length. If *n*K is specified, the maximum length is *n**1,024 (with a maximum value for *n* of 2,097,152). If *n*M is specified, the maximum length is *n**1,048,576 (with a maximum value for *n* of 2,048). If *n*G is specified, the maximum length is *n**1,073,741,824 (with a maximum value for *n* of 2). A CLOB is used to store large single-byte character set (SBCS) character-based data or mixed (multibyte character set (MBCS) and SBCS) character-based data.

Data types (continued)

A *graphic string* is a sequence of bytes that represents double-byte character data. Graphic strings include fixed-length graphic strings of type GRAPHIC(*n*) and varying-length graphic strings of type VARGRAPHIC(*n*), LONG VARGRAPHIC, and DBCLOB(*n*). The length of the string is the number of double-byte characters in the sequence.

- **Fixed-length graphic string, GRAPHIC(*n*):** A fixed-length graphic string is between 1 and 127 double-byte characters long. If a length is not specified, it is assumed to be 1 double-byte character.

- **Varying-length graphic string, VARGRAPHIC(*n*):** VARGRAPHIC(*n*) type strings are varying-length graphic strings that can be up to 16,336 double-byte characters long.

- **LONG VARGRAPHIC:** LONG VARGRAPHIC type strings are varying-length graphic strings that can be up to 16,350 double-byte characters long.

- **Double-byte character large object string, DBCLOB(*n*[K|M|G]):** A double-byte character large object is a varying-length graphic string of double-byte characters that can be up to 1,073,741,823 characters long. If only *n* is specified, *n* is the maximum length. If *n*K is specified, the maximum length is *n**1,024 (with a maximum value for *n* of 1,048,576). If *n*M is specified, the maximum length is *n**1,048,576 (with a maximum value for *n* of 1,024). If *n*G is specified, the maximum length is *n**1,073,741,823 (with a maximum value for *n* of 1). A DBCLOB is used to store large DBCS (double-byte character set) character-based data.

A *binary string* is a sequence of bytes. Binary strings include varying-length strings of type BLOB(*n*), which are used to hold nontraditional data such as pictures, voice, or mixed media, and which can also hold structured data for user-defined types and user-defined functions.

- **Binary large object, BLOB(*n*[K|M|G]):** A binary large object is a varying-length string that can be up to 2,147,483,647 bytes long. If only *n* is specified, *n* is the maximum length. If *n*K is specified, the maximum length is *n**1,024 (with a maximum value for *n* of 2,097,152). If *n*M is specified, the maximum length is *n**1,048,576 (with a maximum value for *n* of 2,048). If *n*G is specified, the maximum length is *n**1,073,741,824 (with a maximum value for *n* of 2).

Datetime data types include DATE, TIME, and TIMESTAMP. Datetime values can be used in certain arithmetic and string operations, and they are compatible with certain strings, but they are neither strings nor numbers.

- **DATE:** A DATE is a three-part value (year, month, and day). The range of the year part is 0001 to 9999. The range of the month part is 1 to 12. The range of the day part is 1 to *n,* where the value of *n* depends on the month. The length of a DATE column is 10 bytes.

- **TIME:** A TIME is a three-part value (hour, minute, and second). The range of the hour part is 0 to 24. The range of both the minute part and the second part is 0 to 59. If the hour is 24, the minute value and the second value are both zero. The length of a TIME column is 8 bytes.

- **TIMESTAMP:** A TIMESTAMP is a seven-part value (year, month, day, hour, minute, second, and microsecond). The range of the year part is 0001 to 9999. The range of the month part is 1 to 12. The range of the day part is 1 to *n,* where the value of *n* depends on the month. The range of the hour part is 0 to 24. The range of both the minute part and the second part is 0 to 59. The range of the microsecond part is 000000 to 999999. If the hour is 24, the minute value, the second value, and the microsecond value are all zero. The length of a TIMESTAMP column is 26 bytes.

String representations of datetime values: Although internal representations of DATE, TIME, and TIMESTAMP values are transparent to the user, dates, times, and timestamps can be represented by character strings, and the CHAR scalar function (see The parts of speech of SQL on page 4) can be used to create a string representation of a datetime value.

- A string representation of a *date* value is a character string that starts with a digit and has a length of at least eight characters. Leading zeros can be omitted from the month and the day parts of the date value.

- A string representation of a *time* value is a character string that starts with a digit and has a length of at least four characters. A leading zero can be omitted from the hour part of the time value, and seconds can be omitted entirely. If a value for seconds is not specified, it is assumed to be 0.
- A string representation of a *timestamp* value is a character string that starts with a digit and has a length of at least 16 characters. The complete string representation of a timestamp has the form *yyyy-mm-dd-hh.mm.ss.nnnnnn.* Leading zeros can be omitted from the month, day, or hour part of the timestamp value, and microseconds can be truncated or omitted entirely. If any trailing zeros are omitted from the microseconds part of the timestamp value, a value of 0 is assumed for the missing digits.

Special registers

A *special register* is a storage area that is defined for an application process by the database manager and is used to store information that can be referenced in SQL statements. DB2 Universal Database currently supports the following special registers:

CLIENT ACCTNG	CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
CLIENT APPLNAME	CURRENT PATH
CLIENT USERID	CURRENT QUERY OPTIMIZATION
CLIENT WRKSTNNAME	CURRENT REFRESH AGE
CURRENT DATE	CURRENT SCHEMA
CURRENT DBPARTITIONNUM	CURRENT SERVER
CURRENT DEFAULT TRANSFORM GROUP	CURRENT TIME
CURRENT DEGREE	CURRENT TIMESTAMP
CURRENT EXPLAIN MODE	CURRENT TIMEZONE
CURRENT EXPLAIN SNAPSHOT	USER

To illustrate how useful special registers can be, consider the SALES table, which has a column named Sales_Date. The following statement inserts a new row into the SALES table, setting the sales date to the value of the `CURRENT DATE` special register:

```
INSERT INTO sales (sales_date, sales_person, region, sales)
  VALUES (CURRENT DATE, 'SMITH', 'Manitoba', 100)
```

For more information about the `INSERT` statement, see Using the INSERT statement to add new rows to tables or views on page 24 .

The system catalog

The database manager creates and maintains two sets of system catalog views. All the system catalog views are created when a database is created; they cannot be explicitly created or dropped. The system catalog views are updated during normal operation. Most of the system catalog views cannot be modified using data manipulation language. SELECT privilege on all of the system catalog views is granted to `PUBLIC` by default.

One set of system catalog views belongs to the SYSCAT schema; these views are not directly updatable. A second set of views, representing a subset of views belonging to the SYSCAT schema, contain statistical information that is used by the optimizer. The *optimizer* is a component of the SQL compiler that chooses an access plan for a DML statement by modeling the execution cost of alternative access plans and choosing the one with the lowest estimated cost. Views belonging to the SYSSTAT schema contain some updatable columns.

The system catalog views can be queried to obtain useful information about a database. For example, the following statement uses the `NOT LIKE` predicate to return the name of each user-defined table with an entry in SYSCAT.TABLES, along with the number of columns in each of those tables, and table status (N = normal; C = check pending):

```
SELECT tabname, colcount, status
  FROM syscat.tables
  WHERE tabschema NOT LIKE 'SYS%'
  ORDER BY tabname
```

And here is a partial result set returned by this query:

TABNAME	COLCOUNT	STATUS
CL_SCHED	4	N
DEPARTMENT	5	N
EMP_ACT	6	C

For more information about the `SELECT` statement, see Using the SELECT statement to retrieve data from database tables on page 18 .

Section 3. Data Control Language (DCL)

Authorization

DCL is a subset of SQL. It is used to provide access control to database objects. There are two levels of security to control access to database objects. The first level, which controls access to the DB2 instance, is managed by the operating system. This level is called *authentication,* and involves verifying a user's identity through a valid user ID and password. The second level of security controls access to a database on the server.

To access a database on the server, you must establish a connection between your DB2 client and the database using the SQL `CONNECT` statement. The syntax of this statement allows you to specify a user ID and password, which DB2 will use to authenticate you. You can also request a password change by supplying your user ID, old password, and new password twice. For example:

```
CONNECT TO sample USER shaman USING mypassword NEW newpassword CONFIRM newpassword
```

If you don't specify a user ID and password with the `CONNECT` statement, DB2 may use the ID and password you logged on with at the client for authentication, or it may prompt you to supply a user ID and password.

The following message tells you that you have made a successful connection:

```
  Database Connection Information

Database server        = DB2/NT 8.1.0
SQL authorization ID   = SHAMAN
Local database alias   = SAMPLE
```

Authorities and privileges

After you are connected, you can access the database, its objects, and the data -- that is, if you have the authority or necessary privileges to do so.

Authorities are broad categories of user rights. They include:

- **SYSADM:** This authority level gives a user the ability to run utilities, issue database and database manager commands, and control database objects throughout the database manager instance. In addition to all of the capabilities granted to the other authorities, users with SYSADM authority can migrate a database, modify the database manager configuration file, and grant DBADM authority.

- **SYSCTRL:** This authority level enables a user to perform all administrative tasks, such as running maintenance and utility operations against the database manager instance and its databases. A user with SYSCTRL authority cannot access data directly unless specifically granted additional privileges. A user with SYSCTRL authority or higher can perform all functions permitted to users with SYSMAINT authority, and can also create or drop a database; force applications; restore to a new database; create, drop, or alter a tablespace; and update a database, node, or distributed connection services (DCS) directory.

- **SYSMAINT:** A user with this authority level can perform maintenance activities, but not access data within the database instance. A user with SYSMAINT or higher authority can update database configuration files, back up a database or tablespace, restore to an existing database, restore a tablespace, perform roll-forward recovery, stop or start a database instance, run traces, and take system monitor snapshots of an instance or its databases.
- **DBADM:** This authority level applies to individual databases. A user with DBADM authority on a database can perform any administrative task on that database, such as creating database objects, loading data, and monitoring database activity. A user with DBADM authority can read log files, create, activate, and drop event monitors, query the state of a tablespace, update history files, quiesce a tablespace, reorganize a table, and collect catalog statistics using the runstats utility. The creator of a database automatically has DBADM authority on that database.
- **LOAD:** This authority level enables a user to perform load operations on a particular database. If a user only has LOAD authority, the user must also have table-level privileges, such as the INSERT privilege, on a table to be able to load data into that table. If a load operation is to replace existing data in a table, the user must also have the DELETE privilege on the table.

Privileges are specific rights that can be granted to users, allowing them to work with specific objects in the database. A privilege is the right to create or access a database object. Objects on which privileges can be held include databases, schemas, tablespaces, tables, views, nicknames, servers, packages, and indexes.

If you create an object, you have full access to that object. This is known as having CONTROL privilege on the object. A user with CONTROL privilege on an object can let other users have access to the object, and can give other users permission to grant privileges on the object. Privileges can be granted or revoked using the SQL `GRANT` or `REVOKE` statement.

Individual privileges, such as SELECT, INSERT, DELETE, and UPDATE, allow a user to perform a specific function, sometimes on a specific object.

To grant privileges on database objects, you must have SYSADM authority, DBADM authority, CONTROL privilege, or have the `WITH GRANT OPTION` (a selectable option on the `GRANT` statement) on that object. You must have SYSADM or DBADM authority to grant CONTROL privilege to another user. You must have SYSADM authority to grant DBADM authority.

For more extensive treatment of DB2 security, see *the second tutorial in this series*.

Schemas

A *schema* is a collection of named objects, such as tables, views, triggers, and functions. Schemas provide a logical classification of objects in the database. A schema name is used as the first part of a two-part object name. Consider for example the name SMITH.STAFF. In this example, the *fully qualified* name of the STAFF table includes the schema name, SMITH, to distinguish it from any other table named STAFF in the system catalog.

The schema itself is a database object. A schema can be explicitly created using the `CREATE SCHEMA` statement; it can also be implicitly created when another object is created, if the user creating the object has IMPLICIT_SCHEMA privilege.

When a database is created, all users have IMPLICIT_SCHEMA privilege. This allows any user to create objects in any schema not already in existence. An implicitly created schema allows any user to create other objects in this schema.

Schemas have privileges associated with them. This allows the schema owner to control which users have the privilege to create, alter, and drop objects in the schema. A schema owner is initially given all of these privileges on the schema, with the ability to grant them to others. An implicitly created schema is owned by the system, and all users are initially given the privilege to create objects in that schema. A user with SYSADM or DBADM authority can change the privileges held by users on any schema, even one that was implicitly created.

Authorization IDs and authorization names

An *authorization ID* is a character string that is obtained by the database manager when a connection is established between the database manager and an application process. It designates a set of privileges. Authorization IDs are used by the database manager to provide authorization checking of SQL statements; an authorization ID applies to every SQL statement.

An *authorization name* specified in an SQL statement is simply an identifier that is used within that statement. For example, an authorization name is used in a `CREATE SCHEMA` statement to designate the owner of the schema, and an authorization name is used in a `GRANT` or `REVOKE` statement to designate the target of a grant or the revoke operation.

Using the GRANT statement

Authorities and privileges can be explicitly granted to an individual user or to a group through invocation of the GRANT statement. Several flavors of the GRANT statement are available; the particular one that you'll use will depend on the object on which privileges are being granted. Objects on which privileges can be granted include databases, tablespaces, tables, views, indexes, packages, and schemas (see Schemas on page 12). In general, the GRANT statement looks like this:

```
GRANT privilege ON object-type object-name
  TO [{USER | GROUP | PUBLIC}] authorization-name
  [WITH GRANT OPTION]
```

Specifying the keyword PUBLIC grants the privilege to all users. Some operating systems allow users and groups to have the same name. In such cases, you can specify the optional keyword USER or GROUP to distinguish them. For example:

```
GRANT INSERT, DELETE ON TABLE staff TO USER rosita WITH GRANT OPTION
```

In this example, the WITH GRANT OPTION gives Rosita the ability to grant the INSERT or DELETE privilege to other users. Of course, if another user already has SYSADM authority, DBADM authority on the database that contains the STAFF table, or CONTROL privilege on the STAFF table, that user already has the authority to grant the INSERT or the DELETE privilege on the STAFF table.

Using the REVOKE statement

Authorities and privileges can be explicitly revoked from an individual user or a group through invocation of the REVOKE statement. As with the GRANT statement, several flavors of the REVOKE statement are available, depending on the object from which privileges are being revoked. Objects from which privileges can be revoked include databases, tablespaces, tables, views, indexes, packages, and schemas. In general, the REVOKE statement looks like this:

```
REVOKE privilege ON object-type object-name
  FROM [{USER | GROUP | PUBLIC}] authorization-name
```

Specifying the keyword PUBLIC revokes the privilege from all users. Some operating systems allow users and groups to have the same name. In such cases, you can specify the optional keyword USER or GROUP to distinguish them.

In the following example, the optional keyword ALL is used to revoke all privileges held by Joanna on the STAFF table:

```
REVOKE ALL PRIVILEGES ON TABLE staff FROM joanna
```

To revoke privileges on an object, you must have SYSADM authority, DBADM authority on the database that contains the object, or CONTROL privilege on the object. Having the WITH GRANT OPTION on an object does not allow you to revoke privileges on that object.

Even if you've revoked privileges on an object from a group, you can't be sure you've revoked privileges from each *member* of that group. If any members of the group had previously been granted privileges as individuals or as members of another group, they retain these privileges unless you issue additional `REVOKE` statements.

Section 4. Data Definition Language (DDL)

Using the CREATE statement to make new database objects

The `CREATE` statement is used to create database objects, including:

- Buffer pools
- Event monitors
- Functions
- Indexes
- Schemas
- Stored procedures
- Tables
- tablespaces
- Triggers
- Views

The system catalog is updated whenever you create a database object.

Consider the `CREATE TABLE` statement. This statement has a large number of options that let you precisely define the kind of table you want to create. In its simplest form, the `CREATE TABLE` statement requires only that you specify one or more columns and associated data types. Consider this example:

```
CREATE TABLE org (
  deptnumb SMALLINT NOT NULL,
  deptname VARCHAR(14),
  manager SMALLINT,
  division VARCHAR(10),
  location VARCHAR(13) )
```

This statement specifies the creation of a five-column table called ORG. The ORG table is actually part of the SAMPLE database that comes with DB2. Each column represents an attribute of the organization. Data pertaining to these attributes can be collected and stored in the table. The first column, DeptNumb, cannot have NULL values, because this column will represent a *unique key* for the table: if each value in this column is unique, a specific value can be used to uniquely identify a row in the table.

Using the DECLARE statement to make new database objects

The `DECLARE` statement is similar to the `CREATE` statement, except that it is used to create temporary tables that exist only for the duration of a database connection. Temporary tables are useful when you are working with intermediate results. Declared tables can be referenced like any other table, and they can be altered or dropped like any other table. A table is the only object that can be declared. The system catalog is *not* updated when you declare a temporary table. You can declare a temporary table by using the `DECLARE GLOBAL TEMPORARY TABLE` statement. For example:

```
DECLARE GLOBAL TEMPORARY TABLE session.temp1
  LIKE employee
  ON COMMIT PRESERVE ROWS
  NOT LOGGED
IN mytempspace
```

In this example, the `DECLARE GLOBAL TEMPORARY TABLE` statement is used to declare a temporary table named TEMP1, located in an existing user temporary tablespace named MYTEMPSPACE. (The user temporary tablespace must already exist before you can issue this statement.) The columns in this table will have the same names and definitions as the columns in the EMPLOYEE table. The rows of the temporary table will be preserved (not deleted) whenever a `COMMIT` statement is processed. Finally, changes to the temporary table are not logged (this is the only option).

Temporary tables must be explicitly (or will be implicitly) qualified by the schema name SESSION, because each session that defines a declared table has its own (possibly unique) description of that temporary table.

Using the ALTER statement to change database objects

The `ALTER` statement is used to change some of the characteristics of an existing database object, including:

- Buffer pools
- Tables
- tablespaces
- Views

You cannot alter an index. If you want to change an index, you must drop it and then create a new one with a different definition.

Consider the following example of an `ALTER TABLE` statement. In this example, a new column is being added to the ORG table that we created in Using the CREATE statement to make new database objects on page 15 . The new State column will eventually contain a two-character state code for each department record.

```
ALTER TABLE org
  ADD state char(2)
```

Using the DROP statement to get rid of database objects

You can drop any object that was created through an SQL `CREATE` or `DECLARE` statement, including:

- Buffer pools
- Event monitors
- Functions
- Indexes
- Schemas
- Stored procedures
- Tables
- tablespaces
- Triggers
- Views

The `DROP` statement removes object definitions from the system catalog and therefore from the database itself. Here is an example of the `DROP TABLE` statement:

```
DROP TABLE org
```

Because database objects can be dependent on other database objects, dropping an object can result in a related object becoming invalid.

Section 5. Data Manipulation Language (DML)

Using the SELECT statement to retrieve data from database tables

The `SELECT` statement is used to retrieve table or view data. In its simplest form, the `SELECT` statement can be used to retrieve all the data in a table. For example, to retrieve all the STAFF data from the SAMPLE database, issue the following command:

```
SELECT * FROM staff
```

Here is a partial result set returned by this query:

ID	NAME	DEPT	JOB	YEARS	SALARY	COMM
10	Sanders	20	Mgr	7	18357.50	-
20	Pernal	20	Sales	8	18171.25	612.45
30	Marenghi	38	Mgr	5	17506.75	-

To restrict the number of rows in a result set, use the `FETCH FIRST` clause. For example:

```
SELECT * FROM staff FETCH FIRST 10 ROWS ONLY
```

You can retrieve specific columns from a table by specifying a *select list* of column names separated by commas. For example:

```
SELECT name, salary FROM staff
```

Use the `DISTINCT` clause to eliminate duplicate rows in a result set. For example:

```
SELECT DISTINCT dept, job FROM staff
```

Use the AS clause to assign a meaningful name to an expression or an item in the select list. For example:

```
SELECT name, salary + comm AS pay FROM staff
```

Without the AS clause, the derived column would have been named 2, indicating that it is the second column in the result set.

Using the WHERE clause and predicates to limit the

amount of data returned by a query

Use the `WHERE` clause to select specific rows from a table or view by specifying one or more selection criteria, or search conditions. A *search condition* consists of one or more predicates. A predicate specifies something about a row that is either true or false (see The parts of speech of SQL on page 4). When building search conditions, be sure to:

- Apply arithmetic operations only to numeric data types
- Make comparisons only among compatible data types
- Enclose character values within single quotation marks
- Specify character values exactly as they appear in the database

Let's look at some examples.

- Find the names of staff members whose salaries are greater than $20,000:

  ```
  "SELECT name, salary FROM staff
    WHERE salary > 20000"
  ```

  Enclosing the statement within double quotation marks keeps your operating system from misinterpreting special characters, such as *>; the greater-than symbol could be interpreted as an output redirection request.

- List the name, job title, and salary of staff members who are not managers and whose salary is greater than $20,000:

  ```
  "SELECT name, job, salary FROM staff
    WHERE job <> 'Mgr'
    AND salary > 20000"
  ```

- Find all names that start with the letter S:

  ```
  SELECT name FROM staff
    WHERE name LIKE 'S%'
  ```

  In this example, the percent sign (%) is a wild card character that represents a string of zero or more characters.

A *subquery* is a `SELECT` statement that appears within the `WHERE` clause of a main query, and feeds its result set to that `WHERE` clause. For example:

```
"SELECT lastname FROM employee
  WHERE lastname IN
  (SELECT sales_person FROM sales
    WHERE sales_date < '01/01/1996')"
```

A *correlation name* is defined in the `FROM` clause of a query, and can serve as a convenient short name for a table. Correlation names also eliminate ambiguous references to identical column names from different tables. For example:

```
"SELECT e.salary FROM employee e
  WHERE e.salary <
  (SELECT AVG(s.salary) FROM staff s)"
```

Using the ORDER BY clause to sort results

Use the `ORDER BY` clause to sort the result set by values in one or more columns. The column names that are specified in the `ORDER BY` clause do not have to be specified in the select list. For example:

```
"SELECT name, salary FROM staff
  WHERE salary > 20000
  ORDER BY salary"
```

You can sort the result set in descending order by specifying `DESC` in the `ORDER BY` clause:

```
ORDER BY salary DESC
```

Using joins to retrieve data from more than one table

A *join* is a query that combines data from two or more tables. It is often necessary to select information from two or more tables, since needed data is often distributed. A join adds columns to the result set. For example, a full join of two three-column tables produces a result set with six columns.

The simplest join is one in which there are no specified conditions. For example:

```
SELECT deptnumb, deptname, manager, id, name, dept, job
  FROM org, staff
```

This statement returns all combinations of rows from the ORG table and the STAFF table. The first three columns come from the ORG tables, and the last four columns come from the STAFF table. Such a result set (the *cross product* of the two tables) is not very useful. What is needed is a *join condition* to refine the result set. For example, here is a query that is designed to identify staff members who are managers:

```
SELECT deptnumb, deptname, id AS manager_id, name AS manager
```

```
  FROM org, staff
  WHERE manager = id
  ORDER BY deptnumb
```

And here is a partial result set returned by this query:

DEPTNUMB	DEPTNAME
10	Head Office
15	New England
20	Mid Atlantic

Using joins to retrieve data from more than one table (continued)

The statement we looked at in the last panel is an example of an inner join. *Inner joins* return only rows from the cross product that meet the join condition. If a row exists in one table but not the other, it is not included in the result set. To explicitly specify an inner join, we can rewrite the previous query with an `INNER JOIN` operator in the `FROM` clause:

```
...
  FROM org INNER JOIN staff
  ON manager = id
...
```

The keyword `ON` specifies the join conditions for the tables being joined. Remember, DeptNumb and DeptName are columns in the ORG table, and Manager_ID and Manager are based on columns (ID and Name) in the STAFF table. The result set for the inner join consists of rows that have matching values for the Manager and ID columns in the *left table* (ORG) and the *right table* (STAFF), respectively. (When you perform a join on two tables, you arbitrarily designate one table to be the left table and the other to be the right.)

Outer joins return rows that are generated by an inner join operation, plus rows that would not be returned by the inner join operation. There are three types of outer joins:

- A *left outer join* includes the inner join *plus* the rows from the *left* table that are not returned by the inner join. This type of join uses the `LEFT OUTER JOIN` (or `LEFT JOIN`) operator in the `FROM` clause.

- A *right outer join* includes the inner join *plus* the rows from the *right* table that are not returned by the inner join. This type of join uses the `RIGHT OUTER JOIN` (or `RIGHT JOIN`) operator in the `FROM` clause.

- A *full outer join* includes the inner join *plus* the rows from *both the left table and the right table* that are not returned by the inner join. This type of join uses the `FULL OUTER JOIN` (or `FULL JOIN`) operator in the `FROM` clause.

More complex queries can be constructed to answer more difficult questions. The following query is designed to generate a list of employees who are responsible for projects, identifying those employees who are also managers by listing the departments that they manage:

```
SELECT empno, deptname, projname
  FROM (employee
  LEFT OUTER JOIN project
  ON respemp = empno)
  LEFT OUTER JOIN department
  ON mgrno = empno
```

The first outer join gets the name of any project for which the employee is responsible; this outer join is enclosed by parentheses and is resolved first. The second outer join gets the name of the employee's department if that employee is a manager.

Using set operators to combine two or more queries into a single query

You can combine two or more queries into a single query by using the `UNION`, `EXCEPT`, or `INTERSECT` set operator. Set operators process the results of the queries, eliminate duplicates, and return the final result set.

- The `UNION` set operator generates a result table by combining two or more other result tables.
- The `EXCEPT` set operator generates a result table by including all rows that are returned by the first query, but not by the second or any subsequent queries.
- The `INTERSECT` set operator generates a result table by including only rows that are returned by all the queries.

Following is an example of a query that makes use of the `UNION` set operator. The same query could use the `EXCEPT` or the `INTERSECT` set operator by substituting the appropriate keyword for `UNION`.

```
"SELECT sales_person FROM sales
  WHERE region = 'Ontario-South'
UNION
SELECT sales_person FROM sales
  WHERE sales > 3"
```

Using the GROUP BY clause to summarize results

Use the `GROUP BY` clause to organize rows in a result set. Each group is represented by a single row in the result set. For example:

```
SELECT sales_date, MAX(sales) AS max_sales FROM sales
  GROUP BY sales_date
```

This statement returns a list of sales dates from the SALES table. The SALES table in the SAMPLE database contains sales data, including the number of successful transactions by a particular sales person on a particular date. There is typically more than one record per date. The `GROUP BY` clause groups the data by sales date, and the `MAX` function (see Using functions to transform data on page 24) in this example returns the maximum number of sales recorded for each sales date.

A different flavor of the `GROUP BY` clause includes the specification of the `GROUPING SETS` clause. *Grouping sets* can be used to analyze data at different levels of aggregation in a single pass. For example:

```
SELECT YEAR(sales_date) AS year, region, SUM(sales) AS tot_sales
  FROM sales
  GROUP BY GROUPING SETS (YEAR(sales_date), region, () )
```

Here, the `YEAR` function is used to return the year portion of date values, and the `SUM` function is used to return the total in each set of grouped sales figures. The *grouping sets list* specifies how the data is to be grouped, or *aggregated.* A pair of empty parentheses is added to the grouping sets list to get a grand total in the result set. The statement returns the following:

YEAR	**REGION**
-	-
-	Manitoba
-	Ontario-North
-	Ontario-South
-	Quebec
1995	-
1996	-

A statement that is almost identical to the previous one, but that specifies the `ROLLUP` clause, or the `CUBE` clause instead of the `GROUPING SETS` clause, returns a result set that provides a more detailed perspective on the data, it may, for instance, provide summaries by location or time.

The `HAVING` clause is often used with a `GROUP BY` clause to retrieve results for groups that satisfy only a specific condition. A `HAVING` clause can contain one or more predicates that compare some property of the group with another property of the group or a constant. For example:

```
"SELECT sales_person, SUM(sales) AS total_sales FROM sales
  GROUP BY sales_person
  HAVING SUM(sales) > 25"
```

This statement returns a list of salespeople whose sales totals exceed 25.

Using functions to transform data

A *database function* is a relationship between a set of input data values and a single result value. DB2 Universal Database provides many built-in functions, including column functions and scalar functions:

- *Column functions* operate on a set of values in a column. For example:
  - `SUM(sales)` returns the sum of the values in the Sales column.
  - `AVG(sales)` returns the sum of the values in the Sales column divided by the number of values in that column.
  - `MIN(sales)` returns the smallest value in the Sales column.
  - `MAX(sales)` returns the largest value in the Sales column.
  - `COUNT(sales)` returns the number of non-null values in the Sales column.
- *Scalar functions* operate on a single value to return another single value. For example:
  - `ABS(-5)` returns the absolute value of -5 -- that is, 5.
  - `HEX(69)` returns the hexadecimal representation of the number 69 -- that is, 45000000.
  - `LENGTH('Pierre')` returns the number of bytes in the string "Pierre" -- that is, 6. For a GRAPHIC string, the LENGTH function returns the number of double-byte characters.
  - `YEAR('03/14/2002')` extracts the year portion of 03/14/2002 -- that is, 2002.
  - `MONTH('03/14/2002')` extracts the month portion of 03/14/2002 -- that is, 3.
  - `DAY('03/14/2002')` extracts the day portion of 03/14/2002 -- that is, 14.
  - `LCASE('SHAMAN')` or `LOWER('SHAMAN')` returns a string in which all of the characters have been converted to lowercase characters -- shaman, in this case.
  - `UCASE('shaman')` or `UPPER('shaman')` returns a string in which all of the characters have been converted to uppercase characters -- SHAMAN, in this case.

Using the INSERT statement to add new rows to tables or views

The `INSERT` statement is used to add new rows to a table or a view. Inserting a row into a view also inserts the row into the table on which the view is based. You can:

- Use a VALUES clause to specify column data for one or more rows. For example:

  ```
  INSERT INTO staff VALUES (1212,'Cerny',20,'Sales',3,90000.00,30000.00)

  INSERT INTO staff VALUES (1213,'Wolfrum',20,'Sales',2,90000.00,10000.00)
  ```

  Or the equivalent:

  ```
  INSERT INTO staff (id, name, dept, job, years, salary, comm)
    VALUES
    (1212,'Cerny',20,'Sales',3,90000.00,30000.00),
    (1213,'Wolfrum',20,'Sales',2,90000.00,10000.00)
  ```

- Specify a fullselect to identify data that is to be copied from other tables or views. A *fullselect* is a statement that generates a result table. For example:

  ```
  CREATE TABLE pers LIKE staff

  INSERT INTO pers
    SELECT id, name, dept, job, years, salary, comm
      FROM staff
      WHERE dept = 38
  ```

Using the UPDATE statement to change data in tables or views

The UPDATE statement is used to change the data in a table or a view. You can change the value of one or more columns for each row that satisfies the conditions specified by a WHERE clause. For example:

```
UPDATE staff
  SET dept = 51, salary = 70000
    WHERE id = 750
```

Or the equivalent:

```
UPDATE staff
  SET (dept, salary) = (51, 70000)
    WHERE id = 750
```

If you don't specify a WHERE clause, DB2 updates each row in the table or view!

Using the DELETE statement to get rid of data

The `DELETE` statement is used to delete entire rows of data from a table. You can delete each row that satisfies the conditions specified by a `WHERE` clause. For example:

```
DELETE FROM staff
  WHERE id IN (1212, 1213)
```

If you don't specify a `WHERE` clause, DB2 deletes all the rows in the table or view!

Section 6. SQL procedures

Creating an SQL procedure

An SQL *procedure* is a stored procedure whose body is written in SQL. The body contains the logic of the SQL procedure. It can include variable declarations, condition handling, flow-of-control statements, and DML. Multiple SQL statements can be specified within a *compound statement,* which groups several statements together into an executable block.

An SQL procedure is created when you successfully invoke a `CREATE PROCEDURE (SQL)` statement, which defines the SQL procedure with an application server. SQL procedures are a handy way to define more complex queries or tasks that can be called whenever they are needed. The following steps will create a command-line processor (CLP) script (called `createSQLproc.db2`) that will create a simple SQL procedure:

1. Connect to the SAMPLE database.

2. Issue the following command:

```
db2 -td@ -vf createSQLproc.db2
```

This `db2` command specifies the `-td` option flag, which tells the command-line processor to define and to use `@` as the statement termination character; the `-v` option flag, which tells the command-line processor to echo command text to standard output; and the `--f` option flag, which tells the command-line processor to read command input from the specified file instead of from standard input.

```
CREATE PROCEDURE sales_status
(IN quota INTEGER, OUT sql_state CHAR(5))
DYNAMIC RESULT SETS 1
LANGUAGE SQL
BEGIN
  DECLARE SQLSTATE CHAR(5);
  DECLARE rs CURSOR WITH RETURN FOR
  SELECT sales_person, SUM(sales) AS total_sales
    FROM sales
    GROUP BY sales_person
    HAVING SUM(sales) > quota;
  OPEN rs;
  SET sql_state = SQLSTATE;
END @
```

This procedure, called SALES_STATUS, accepts an input parameter called *quota* and returns an output parameter called *sql_state.* The procedure body consists of a single `SELECT` statement that returns the name and the total sales figures for each salesperson whose total sales exceed the specified quota.

Most SQL procedures will accept at least one input parameter. In our example, the input parameter contains a value *(quota)* that is used in the `SELECT` statement contained in the procedure body.

Many SQL procedures will return at least one output parameter. Our example includes an output parameter *(sql_state)* that is used to report the success or failure of the SQL procedure. DB2 returns an `SQLSTATE` value in response to conditions that could be the result of an SQL statement. Because the returned `SQLCODE` or `SQLSTATE` value pertains to the last SQL statement issued in the procedure body, and accessing the values alters the subsequent values of these variables (because an SQL statement is used to access them), the `SQLCODE` or `SQLSTATE` value should be assigned to and returned through a locally defined variable (such as the *sql_state* variable in our example).

The parameter list for an SQL procedure can specify zero or more parameters, each of which can be one of three possible types:

- `IN` parameters pass an input value to an SQL procedure; this value cannot be modified within the body of the procedure.
- `OUT` parameters return an output value from an SQL procedure.
- `INOUT` parameters pass an input value to an SQL procedure and return an output value from the procedure.

SQL procedures can return zero or more result sets. In our example, the SALES_STATUS procedure returns one result set. This has been done by:

1. Declaring the number of result sets that the SQL procedure will return in the `DYNAMIC RESULT SETS` clause.

2. Declaring a cursor in the procedure body (using the `WITH RETURN FOR` clause) for each result set that will be returned. A *cursor* is a named control structure that is used by an application program to point to a specific row within an ordered set of rows. A cursor is used to retrieve rows from a set.

3. Opening the cursor for each result set that will be returned.

4. Leaving the cursor(s) open when the SQL procedure returns.

Variables must be declared at the beginning of the SQL procedure body. To *declare* a variable, you must assign a unique identifier to and specify an SQL data type for the variable and, optionally, assign an initial value to the variable.

The `SET` clause in our sample SQL procedure is an example of a *flow-of-control* clause. The following flow-of-control statements, structures, and clauses can be used for conditional processing within an SQL procedure body:

- The `CASE` structure selects an execution path based on the evaluation of one or more conditions.

- The `FOR` structure executes a block of code for each row of a table.
- The `GET DIAGNOSTICS` statement returns information about the previous SQL statement into an SQL variable.
- The `GOTO` statement transfers control to a labeled block (a section of one or more statements identified by a unique SQL name followed by a colon).
- The `IF` structure selects an execution path based on the evaluation of conditions. The `ELSEIF` and `ELSE` clauses enable you to branch or to specify a default action if the other conditions are false.
- The `ITERATE` clause passes the flow of control to the beginning of a labeled loop.
- The `LEAVE` clause transfers program control out of a loop or block of code.
- The `LOOP` clause executes a block of code multiple times until a `LEAVE`, `ITERATE`, or `GOTO` statement transfers control outside of the loop.
- The `REPEAT` clause executes a block of code until a specified search condition returns true.
- The `RETURN` clause returns control from the SQL procedure to the caller.
- The `SET` clause assigns a value to an output parameter or SQL variable.
- The `WHILE` clause repeatedly executes a block of code while a specified condition is true.

To successfully create SQL procedures, you must have installed the DB2 Application Development Client on the database server. (See *the first tutorial in this series* for more on the Application Development Client.) DB2 converts SQL procedure statements into an equivalent C application by using embedded SQL; this means that before you can create any SQL procedures, you must also install and configure a supported C compiler on the database server. If your database server runs on a Windows operating system, but the environment variables for the C compiler are not system variables, you must update the `DB2_SQLROUTINE_COMPILER_PATH DB2` registry variable on the database server:

- For Microsoft Visual Studio, update the DB2 registry variable with the path for the vcvars32.bat file.
- For IBM VisualAge for C++, update the DB2 registry variable with the path for the setenv.bat file.

If your database server runs on a UNIX-based system, DB2 generates an executable script file (`$HOME/sqllib/function/routine/sr_cpath`) the first time you compile a stored procedure. This script contains the default values for the compiler environment variables on your operating system.

Calling an SQL procedure

You can use the SQL `CALL` statement to call SQL procedures from the DB2 command line. The procedure being called must be defined in the system catalog. To call the SQL procedure SALES_STATUS (which we outline above; see Creating an SQL procedure on page 27), perform the following steps:

1. Connect to the SAMPLE database.
2. Issue the following statement:

   ```
   db2 CALL sales_status (25, ?)
   ```

   Because parentheses have special meaning to the command shell on UNIX-based systems, on those systems they must be preceded with a backslash (\) character or be enclosed by double quotation marks, as follows:

   ```
   db2 "CALL sales_status (25, ?)"
   ```

   Do not include double quotation marks if you are using the command line processor (CLP) in interactive input mode, characterized by the `db2` => input prompt.

In this example, a value of 25 for the input parameter *quota* is passed to the SQL procedure, as well as a question mark (?) place-holder for the output parameter *sql_state.* The procedure returns the name and the total sales figures for each salesperson whose total sales exceed the specified quota (25). The following is sample output returned by this statement:

```
SQL_STATE: 00000

SALES_PERSON                                TOTAL_SALES
GOUNOT                                      50
LEE                                         91
"SALES_STATUS" RETURN_STATUS: "0"
```

Client applications written in any supported language can call SQL procedures.

Section 7. Summary

Summary

This tutorial was designed to introduce you to Structured Query Language (SQL), and to some of the ways that DB2 Universal Database uses SQL to define database objects and to manipulate data in relational databases. This tutorial has covered the fundamentals of SQL, including SQL language elements, Data Control Language (DCL), Data Definition Language (DDL), Data Manipulation Language (DML), and SQL procedures.

Resources

Good places to find additional information about DB2 Universal Database and SQL are:

- *IBM DB2 Universal Database SQL Reference, Vol. 1, Version 8*, SC09-4844-00; *IBM DB2 Universal Database SQL Reference, Vol. 2, Version 8*, SC09-4845-00. International Business Machines Corporation, 2002.
- *DB2: The Complete Reference*. Melnyk, Roman B., and Paul Z. Zikopoulos, Osborne/McGraw-Hill, 2001.

For more information on the DB2 V8.1 Family Fundamentals Exam 700:

- *IBM Data Management Skills* information
- Download a *self-study course for experienced Database Administrators (DBAs)* to quickly and easily gain skills in DB2 UDB.
- Download a *self study course for experienced relational database programmers* who would like to know more about DB2.
- *General Certification Information*, including some book suggestions, exam objectives, courses

Check out the other parts of the DB2 V8.1 Family Fundamentals Certification Prep series:

- *DB2 V8.1 Family Fundamentals Certification Prep, Part 1 of 6: DB2 Planning*
- *DB2 V8.1 Family Fundamentals Certification Prep, Part 2 of 6: DB2 Security*
- *DB2 V8.1 Family Fundamentals Certification Prep, Part 3 of 6: Accessing DB2 UDB Data*
- *DB2 V8.1 Family Fundamentals Certification Prep, Part 5 of 6: Working with DB2 UDB Objects*
- *DB2 V8.1 Family Fundamentals Certification Prep, Part 6 of 6: Data Concurrency*

Feedback

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT style sheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.